01｜实现一门超简单的语言最快需要多久？

2021-08-09 宫文学

《手把手带你写一门编程语言》 课程介绍 >

讲述：宫文学
时长 16:54 大小 15.48M

你好，我是宫文学。

说到实现一门计算机语言，你肯定觉得这是一个庞大又复杂的工程，工作量巨大！

这个理解，我只能说部分正确。其实，有的时候，实现一门语言的速度也可以很快。比如，当年兰登. 艾克（Brendan Eich）只花了 10 天时间就把 JavaScript 语言设计出来了。当然，语言跟其他软件一样，也需要不断迭代，至今 JS 的标准和实现仍在不停的演化。

如果我说，你也完全可以在这么短的时间内实现一门语言，甚至都不需要那么长时间，你一定会觉得我是在哗众取宠、标题党。

别急，我再补充说明一下，你马上就会认可我的说法了。这个让你一开始实现的版本，只是为了去探索计算机语言的原理，是高度简化的版本，并不要求马上能实用。你可以把它看做是一个原型系统，仅此而已，实现起来不会太复杂。

好吧，我知道你肯定还在心里打鼓：再简单的计算机语言，那也是一门语言呀，难度又能低到哪里去？

这样，先保留你的疑虑，我们进入今天的课程。**今天我就要带你挑战，仅仅只用一节课时间，就实现一门超简洁的语言。**我会暂时忽略很多的技术细节，带你抓住实现一门计算机语言的骨干部分，掌握其核心原理。在这节课中，你会快速获得两个技能：

如何通过编译器来理解某个程序；

如何解释执行这个程序。

这两个点，分别是编译时的核心和运行时的核心。理解了这两个知识点，你就大致理解计算机语言是如何工作的了！

我们的任务

这节课，我们要让下面这个程序运行起来：

复制代码

```
//一个函数的声明，这个函数很简单，只打印"Hello World!"
function sayHello(){
    println("Hello World!");
}
//调用刚才声明的函数
sayHello();
```

这个程序做了两件事：第一件是声明了一个函数，叫做 sayHello；第二件事，就是调用 sayHello() 函数。运行这个程序的时候，我们期待它会输出“Hello World！”。

这个程序看上去还挺像那么回事的，但其实为降低难度，我们对 JavaScript/TypeScript 做了极度的简化：它只支持声明函数和调用函数，在我们的 sayHello() 函数里，它只能调用函数。你可以调用一个自己声明的函数，如 foo，也可以调用语言内置的函数，如示例中的 println()。

这还不够，为了进一步降低难度，我们的编译器是从一个数组里读取程序，而不是读一个文本文件。这个数组的每一个元素是一个单词，分别是 function、sayHello、左括号、右括号等等，这些单词，我们叫它 Token，它们是程序的最小构成单位。注意，最后一个 Token 比较特殊，它叫做 EOF，有时会记做 $，表示程序的结尾。

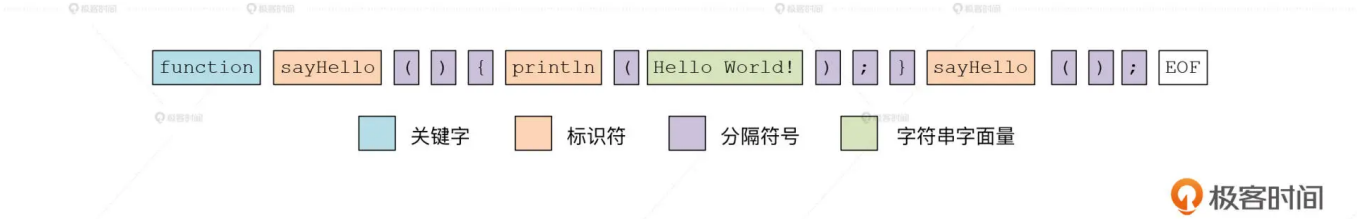

图1：Token串（图中不同的颜色代表不同类型的Token）

好了，现在任务清楚了，那我们开始第一步，解析这个程序。

解析这个程序

解析，英文叫做 Parse，是指读入程序，并形成一个计算机可以理解的数据结构的过程。能够完成解析工作的程序，就叫做解析器（Parser），它是编译器的组成部分之一。

那么，什么数据结构是计算机能够理解的呢？很简单，其实就是一棵树，这棵树叫做**抽象语法树，英文缩写是 AST（Abstract Syntax Tree）**。针对我们的例子，这棵 AST 是下面的样子：

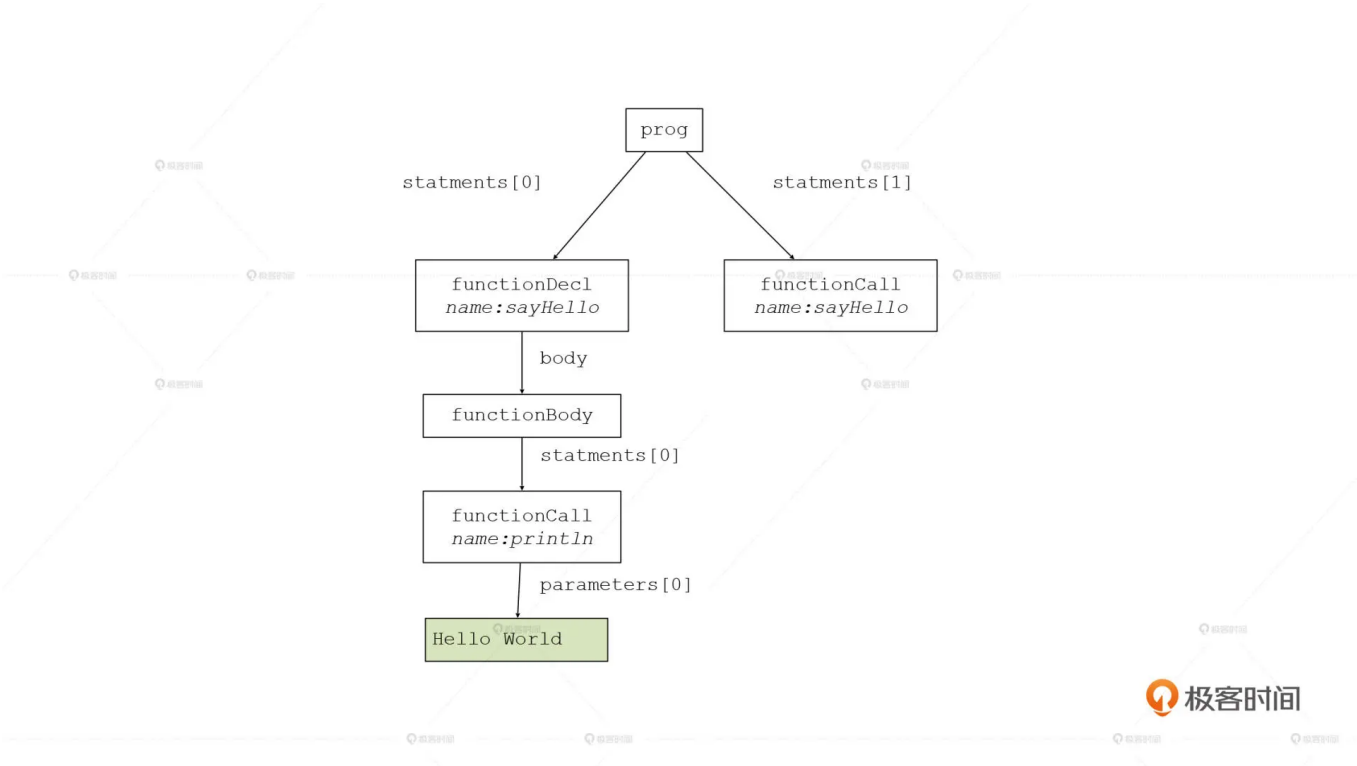

图2：示例代码对应的AST

你仔细看一下这棵树，你会发现它跟我们程序想表达的思想是一样的：

根节点代表了整个程序；

根节点有两个子节点，分别代表一个函数声明和一个函数调用语句；

函数声明节点，又包含了两个子节点，一个是函数名称，一个是函数体；

函数体中又包含一个函数调用语句；

而函数调用语句呢，则是由函数名称和参数列表构成的；

……

通过这样自顶向下的层层分析，你会发现这棵树确实体现了我们原来的程序想表达的意思。其实，这就跟我们自己在阅读文章的时候是一样的，我们的大脑也是把段落分解成句子，再把句子分解成主语、谓语、宾语等一个个语法单元，最终形成一棵树型的结构，我们的大脑是能够理解这种树型结构的。

总结起来，**解析器的工作，就是要读取一个 Token 串，然后把它转换成一棵 AST**。

好了，知道了解析的工作目标后，我们就来实现这个解析器吧！

可是，这怎么下手呢？

你可以琢磨一下，你的大脑是如何理解这些程序，并且把它们在不知不觉之间转化成一棵树的。我们假设，人类的大脑采用了一种自顶向下的分析方式，也就是把一个大的解析任务逐步分解成小的任务，落实到解析器的实现上也是如此。

首先，我们的目的是识别 Token 串的特征，并把它转换成 AST。我们暂时忽略细节，假设我们能够成功地完成这个解析，那么把这个解析动作写成代码，就是：

复制代码

```
1 prog = parseProg();
```

我们再具体一点，看看实现 parseProg() 需要做什么事情。

parseProg() 需要建立程序的子节点，也就是函数声明或者函数调用。我们规定一个程序可以有零到多个函数声明或函数调用。我们把这个语法规则用比较严谨的方法表达出来，是这样的：

```
prog = (functionDecl | functionCall)* ;
```

咦？这个表达方式看上去有点熟悉呀？这个式子的格式叫做 EBNF 格式（扩展巴科斯范式）。你可以看到，它的左边是一个语法成份的名称，右边说的是这个语法成份是由哪些子成分构成的。这样，整个式子就构成了一条语法规则。

不过，编译原理的教科书里，有时会用产生式的格式。EBNF 格式和产生式是等价的，区别是产生式不允许使用？、* 和 + 号，而是用递归来表示多个元素的重复，用ε来表示不生成任何字符串。如果我们把上述语法规则用产生式来表示的话，相当于下面四条：

```
prog -> statement prog
prog -> ε
statement -> functionDecl
statement -> functionCall
```

你马上就能看出来，还是 EBNF 格式更简洁吧？

好了，根据这条语法规则，一个程序要么是由函数声明构成的，要么是由函数调用构成的，那么我们就把它们一一解析出来，变成 prog 的子节点就好了！

可是，我们如何知道接下来遇到的是函数声明还是函数调用呢？

有办法！办法就是：试试看！

什么叫试试看？就是先试试是不是函数声明。如果成功的话，解析器就会返回一棵代表函数声明的子树；如果不成功的话，解析器就会返回 null，然后你可以再试试看是不是函数调用。你可能会问：就这么简单？对，就这么简单。

实现上述逻辑的代码如下：

复制代码

```
parseProg():Prog{
    let stmts: Statement[] = [];
    let stmt: Statement|null|void = null;
    while(true){  //每次循环解析一个语句
        //尝试一下函数声明
        stmt = this.parseFunctionDecl();
        if (stmt != null){
            stmts.push(stmt);
            continue;
        }

        //如果前一个尝试不成功，那么再尝试一下函数调用
        stmt = this.parseFunctionCall();
        if (stmt != null){
            stmts.push(stmt);
            continue;
        }

        //如果都没成功，那就结束
        if (stmt == null){
            break;
        }
    }
    return new Prog(stmts);
}
```

现在我们再往下一层，看看如何解析函数声明。

函数声明包括 function 关键字、函数名称、一对括号和函数体。我们依然将它表达成语法规则，是这样的：

复制代码

```
functionDecl: "function" Identifier "(" ")"  functionBody;
```

注意，在这个新的语法规则里，我们发现了“function”、“Identifier”这样的元素，这些元素是不能再展开的，我们把它们叫做终结符。你会发现，终结符都是从 Token 转化而来的。而像“functionBody”这些用小写字母开头的元素，还可以再继续展开，所以叫做非终结符。

那这该怎么解析呢？

还是得挨个去试。先试一下第一个 Token 是不是“function”，再试下第二个 Token 是不是一个合法的标识符，接着试第三个 Token 是不是左括号，依此类推就行了。

其实，在函数声明的语法规则里，像“function”、“Identifier”这些，都已经是最终的 Token 了，所以直接把它们从 Token 串中取出来就行了，其中“Identifier”变成了 AST 中的函数名称。

而函数声明中的另一部分，也就是函数体，是一个比较复杂的结构，所以我们将它拆出来单独定义。体现在 AST 中，它就是函数声明的子节点。这里，我索性把剩余的有关函数体、函数调用的语法规则都写出来。

复制代码

```
functionBody : '{' functionCall* '}' ;
functionCall : Identifier '(' parameterList? ')' ;
parameterList : StringLiteral (',' StringLiteral)* ;
```

解析上述语法规则对应的程序如下：

复制代码

```
/**
     * 解析函数声明
     * 语法规则：
     * functionDecl: "function" Identifier "(" ")"  functionBody;
     */
    parseFunctionDecl():FunctionDecl|null|void{
        let oldPos:number = this.tokenizer.position();
        let t:Token = this.tokenizer.next();
        if (t.kind == TokenKind.Keyword && t.text == "function"){
            t = this.tokenizer.next();
            if (t.kind == TokenKind.Identifier){
                //读取"("和")"
                let t1 = this.tokenizer.next();
                if (t1.text=="("){
                    let t2 = this.tokenizer.next();
                    if (t2.text==")"){
                        let functionBody = this.parseFunctionBody();
                        if (functionBody != null){
                            //如果解析成功，从这里返回
                            return new FunctionDecl(t.text, functionBody);
```

```
                        }
                    }
                    else{
                        console.log("Expecting ')' in FunctionDecl, while we g
                        return;
                    }
                }
                else{
                    console.log("Expecting '(' in FunctionDecl, while we got a
                    return;
                }
            }
        }

        //如果解析不成功，回溯，返回null。
        this.tokenizer.traceBack(oldPos);
        return null;
    }

    /**
     * 解析函数体
     * 语法规则：
     * functionBody : '{' functionCall* '}' ;
     */
    parseFunctionBody():FunctionBody|null|void{
        let oldPos:number = this.tokenizer.position();
        let stmts:FunctionCall[] = [];
        let t:Token = this.tokenizer.next();
        if(t.text == "{"){
            let functionCall = this.parseFunctionCall();
            while(functionCall != null){  //解析函数体
                stmts.push(functionCall);
                functionCall = this.parseFunctionCall();
            }
            t = this.tokenizer.next();
            if (t.text == "}"){
                return new FunctionBody(stmts);
            }
            else{
                console.log("Expecting '}' in FunctionBody, while we got a " +
                return;
            }
        }
        else{
            console.log("Expecting '{' in FunctionBody, while we got a " + t.t
            return;
        }

        //如果解析不成功，回溯，返回null。
        this.tokenizer.traceBack(oldPos);
        return null;
    }
```

```
73
74      /**
75       * 解析函数调用
76       * 语法规则：
77       * functionCall : Identifier '(' parameterList? ')' ;
78       * parameterList : StringLiteral (',' StringLiteral)* ;
79       */
80      parseFunctionCall():FunctionCall|null|void{
81          let oldPos:number = this.tokenizer.position();
82          let params:string[] = [];
83          let t:Token = this.tokenizer.next();
84          if(t.kind == TokenKind.Identifier){
85              let t1:Token = this.tokenizer.next();
86              if (t1.text == "("){
87                  let t2:Token = this.tokenizer.next();
88                  //循环，读出所有
89                  while(t2.text != ")"){
90                      if (t2.kind == TokenKind.StringLiteral){
91                          params.push(t2.text);
92                      }
93                      else{
94                          console.log("Expecting parameter in FunctionCall, whil
95                          return;  //出错时，就不在错误处回溯了。
96                      }
97                      t2 = this.tokenizer.next();
98                      if (t2.text != ")"){
99                          if (t2.text == ","){
100                             t2 = this.tokenizer.next();
101                         }
102                         else{
103                             console.log("Expecting a comma in FunctionCall, wh
104                             return;
105                         }
106                     }
107                 }
108                 //消化掉一个分号：;
109                 t2 = this.tokenizer.next();
110                 if (t2.text == ";"){
111                     return new FunctionCall(t.text, params);
112                 }
113                 else{
114                     console.log("Expecting a comma in FunctionCall, while we g
115                     return;
116                 }
117             }
118         }
119
120         //如果解析不成功，回溯，返回null。
121         this.tokenizer.traceBack(oldPos);
122         return null;
123     }
124 }
```

到这里你会发现，我们做语法分析的思路很简单：

首先，写出语法规则，比如说用 EBNF 格式；

第二，根据语法规则，分别匹配每个子元素。如果这个语法规则中用到了另一个语法规则，就像函数声明里用到了函数体，那么我们就需要递归地匹配这条语法规则。这个层层下降的匹配过程，叫做“**递归下降（Recursive Descent）**”，这就是我们刚刚采用的算法，Java、V8 等很多编译器也都采用了这个递归下降算法。

第三，如果一条语法规则可能有多个展开方式，就像程序里面既可以是函数声明，也可以是函数调用，**那么我们就要依次尝试去匹配。**如果尝试失败，就回退，再去尝试另一个展开方式。

好了，你也知道了，我们刚才用到的方法就是大名鼎鼎的“递归下降算法”。当然，这是它的初级版本，会存在一些缺陷，这个我们后面会讲。不过，即使是像 JDK 里的 Java 编译器、V8 的 JavaScript 编译器、Go 语言的编译器这些成熟的编译器，采用的都是递归下降算法，所以你应该足够重视它。

做一点简单的语义分析

好了，现在我们已经完成了语法分析工作，并形成了 AST。但是，在解释执行这个程序之前，我们还要做一点微小的工作：就是**引用消解（Refrence Resolving）**。

这是什么意思呢？

在我们的程序中有两处函数调用：一处是 println()，我们现在把它看做内置函数就好了，而另一处，是调用我们自定义的函数 sayHello()。那么**引用消解，就是把函数的调用和函数的定义关联到一起**，否则，程序就没法正常运行。由于我们目前的语言很简单，所以引用消解工作也很简单，你可以看看下面的参考代码：

复制代码

```
//遍历prog中的所有语句，找到一个名称匹配的函数声明
private findFunctionDecl(prog:Prog, name:string):FunctionDecl|null{
    for(let x of prog?.stmts){
        let functionDecl = x as FunctionDecl;
        if (typeof functionDecl.body === 'object' &&
            functionDecl.name == name){
            return functionDecl;
        }
```

```
 9      }
10      return null;
11  }
```

引用消解是语义分析过程中必须要做的一项工作，其他更多的语义分析工作我们仍然可以忽略。做完了语义分析以后，AST 上增加了一些信息：

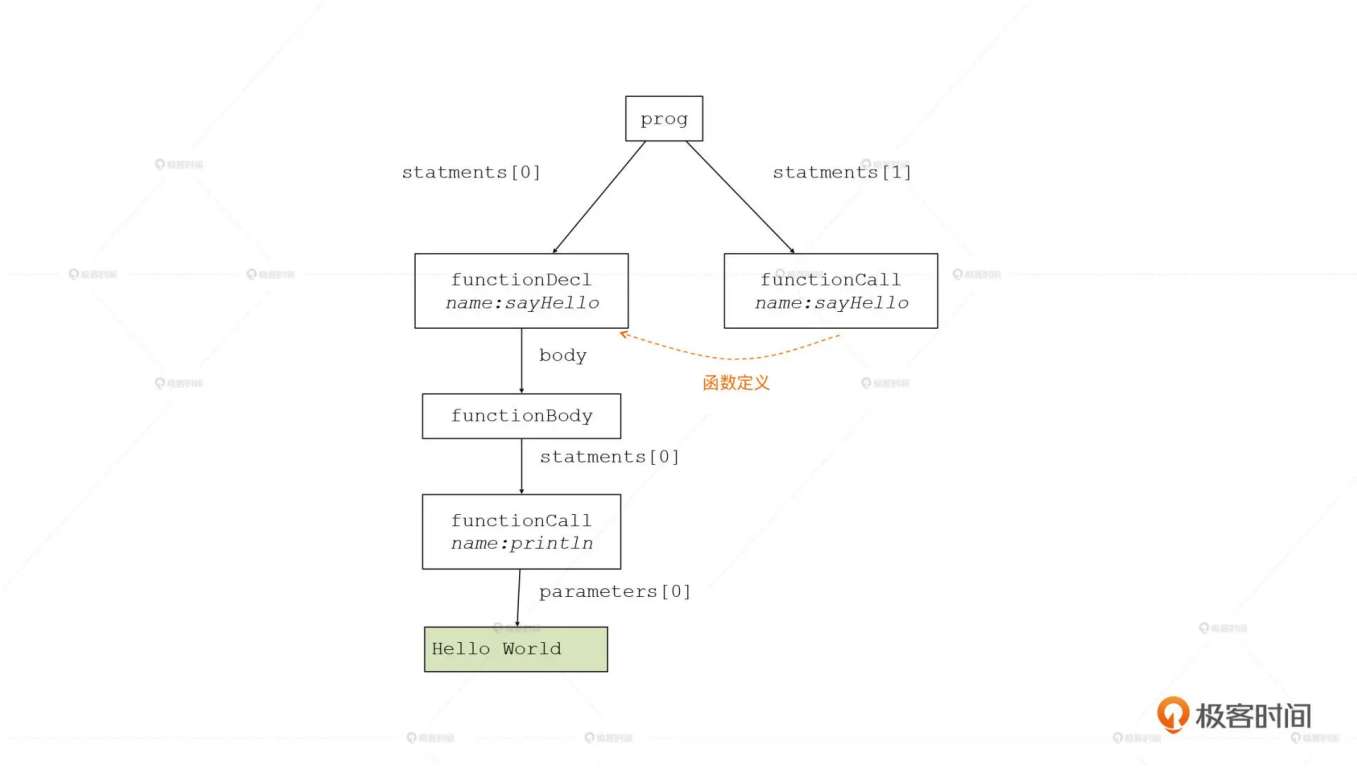

图3：在AST增加了引用信息，可以找到函数的定义

在上图中，你可以看到对 sayHello() 函数做调用的 AST 节点，指向了 sayHello 的函数声明节点。这样，在解释器里执行 sayHello() 的时候，就能找到该执行什么代码了！

做一个简单的解释器来运行这个程序

好了，完成好上面各项准备工作，接下来要实现解释器了。

解释器怎么运行呢？很简单，自顶向下、深度优先地遍历一下 AST 就行了。

具体过程是这样的：

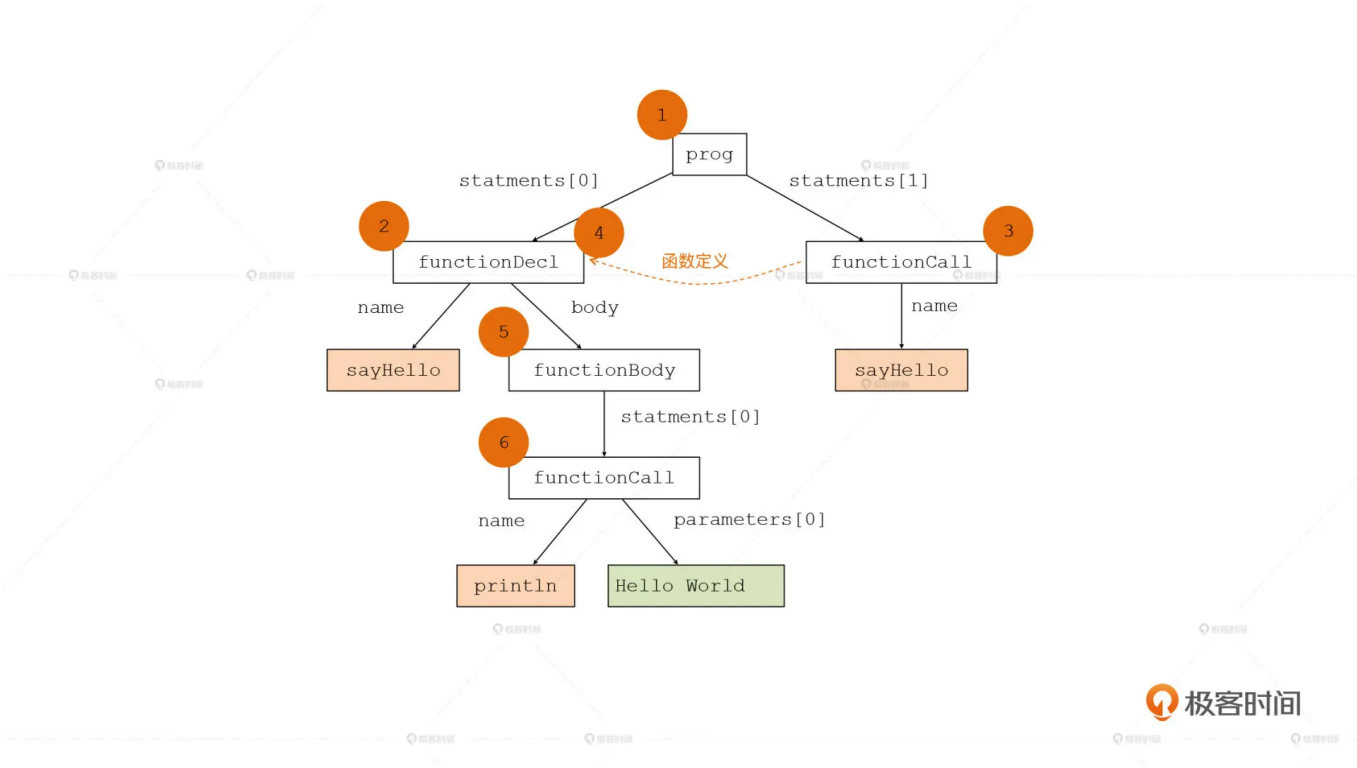

图4：通过遍历AST来解释执行程序

第 1 步，我们要来访问 prog 节点。

第 2 步，接着访问 prog 的第一个子节点的，但这个子节点是一个函数声明，函数声明是不能直接运行的，所以此时访问这个节点不会产生任何动作。

第 3 步，访问 prog 的第二个子节点。这个子节点是一个函数调用，由于我们已经做了引用消解，我们已经知道这个函数在 AST 中的位置了。

第 4 步，我们要跳到函数定义的位置去执行这个函数，然后在第 5 步中执行这个函数的函数体。

第 6 步，我们再执行函数体的下一级节点，也就是 println（“Hello World”）。这实际上是调用一个内置函数，因此我们把“Hello World”作为参数传递给内置函数就行了。

到此，程序执行完毕，输出了“Hello World”。

为了加深你的理解，我再举一个例子。

假如我们的语言现在已经支持表达式了，那么对于 2+3*5 这样一个表达式，解析器也会自动形成一棵 AST。这时，对表达式求值的过程，也就是遍历 AST 的过程。

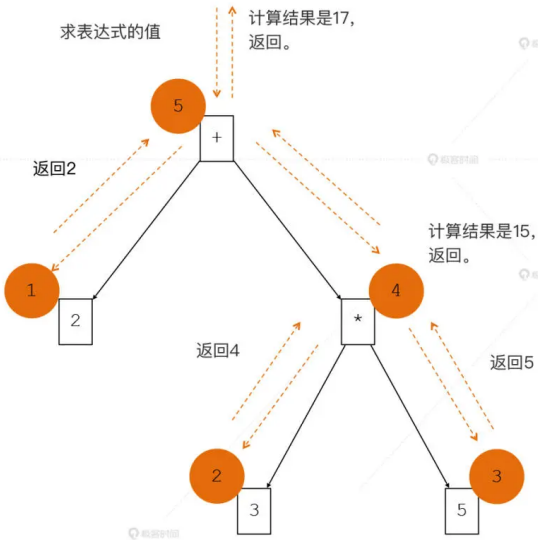

图5：通过遍历AST来计算表达式的值（图中标号，是完全访问完毕一棵子树的顺序）

课程小结

好了，这就是我们今天的全部内容了。虽然我们做了很多的简化，但我们毕竟已经能让一个程序运行起来了。你还可以把示例程序改写一下，比如多写几个自定义的函数，多做几个函数调用，打印出调用过程，等等。

例如，在下面的示例程序中，我又新声明了一个函数 bar，并在 bar 中调用 foo，检验一下多级函数调用是否会正常运行。

复制代码

```
function foo(){
  println("in foo...");
}

function bar(){
  println("in bar...");
  foo();
}

bar();
```

你看，虽然我们的程序特性很少，但毕竟能玩起来了。你只要遵守语言的特性，编写出来的任意程序都是能正确执行的，这就是计算机语言最有魅力的地方！而且，你也可以亲手实现。

我给你把这节课的要点总结一下：

> 要实现一门计算机语言，首先要有一个编译器来把程序转化成计算机可理解的数据（这里是 AST），然后需要有一个解释器来执行它。
>
> 编译器有很多功能，词法分析功能在这讲被我们故意跳过了。在语法分析部分，我们了解了如何用 EBNF 格式来表达语法规则，并初步介绍了递归下降算法；在语义分析部分，我们做了函数的引用消解。
>
> 我们当前版本的解释器，是通过遍历 AST 来运行程序的，方便你理解原理。在后面的课程中，我们会把 AST 转换成更适合解释执行的中间代码（Intermediate Representation，IR），就像 Java 的字节码一样，然后我们再实现一个新的解释器。

这节课我们忽略了很多技术细节，别担心，之后的课程，我们会一一补上这些知识点，让你平滑地学到实现一门语言的所有技能。

思考题

我们今天的课程就到这里了，我给你布置了两个思考题来巩固下今天学习的内容：

问题 1：在这节课讲述的语法规则中，我们做了一些简化，比如在函数声明的时候，我们并没有管参数。如果加上参数，你会怎样改写一下语法规则呢？另外，我们目前用的还是 JavaScript 的语法，如果改成 TypeScript 的语法，带上类型声明，语法规则又会是什么样子呢？你可以练习一下。

问题 2：在今天的课程里，我们的语法分析的算法是“递归下降”算法。不知道你有没有发现，我们的程序里并没有出现函数的递归调用呀，为什么还要说它是递归的呢？

感谢你和我一起学习，如果你觉得我这节课讲得还不错，也欢迎你把它分享给更多对编程语言感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

这节课的示例代码在这里！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 3　提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　课前热身 | 开始学习之前我们要准备什么？

下一篇　02 | 词法分析：识别Token也可以很简单吗？

精选留言 (7)

写留言

青玉白露
2021-08-09

这门课我追定了！宫老师加紧更新哈，内容很赞

展开

作者回复: 多谢支持！

3

全国第一菜
2021-08-10

偶然间习得宫老师的前两门课，如获至宝，对个人提升显著。今天看到开新课了，第一时间来支持，因为知道自己又要变强了！！加油

展开

作者回复: 谢谢肯定，加油！

1

张贺
2021-08-11

代码中RefResolver中的visitFunctionBody方法不应该return吧

_____R
2021-08-11

宫老师，我之前看过一点点 ts，现在主学 go，今天刚把 01 的代码用 go 重写，可以提交到码云上吗

作者回复: 可以呀。你建一个自己的仓库，可以把链接分享出来！

qinsi
2021-08-10

if嵌套看着脑阔疼，建议扩充token类型，增加一些helper方法：

```typescript
/**
 * 解析函数调用 ...
```

展开

作者回复: 没问题，你可以修改优化一下。
事实上，在后面的章节中，这些代码的结构就修改了，变成了直线式的代码，同时增加了语法错误的处理能力。
这一节的代码，是为了尽量避免太多功能，尽量避免复杂化。

千无
2021-08-10

很赞，课后习题一定要做，这才是这门课的精髓，实践它掌握它，课就超值

作者回复: 非常好，这门课就是提倡动手实践的！

P A N ...
2021-08-10

太牛了！

展开